Cloud Computing Project - Lego Backend System

Coumba Louise Mbodji SOW

November 11, 2025

1 Introduction

The goal of this project is to design and implement a scalable backend for a Lego collection management system. The system enables users to manage their Lego sets, create auctions, and interact through comments and bids. The backend leverages a comprehensive set of Azure services: App Service for robust API hosting, Blob Storage for media assets, and Cosmos DB for highly available, low-latency data access. Crucially, Azure Functions are integrated to handle essential event-driven and asynchronous processes, such as the scheduled tasks to close expired auctions and clean up older auction data. This project offers hands-on experience in deploying and operating a functional backend on Azure, emphasizing the critical role of cloud computing in ensuring the scalability, efficiency, and reliability required for managing large-scale, interactive applications.

2 Design

2.1 System Architecture

In this section, we describe the core data models that represent the main components of the system. These models form the basis of the backend and are essential for managing users, Lego sets, auctions, and bids. The data models are represented as classes and are directly associated with the system's operations.

The key components of the system include:

- **User**: Manages user-related information, including name, password, photo, and a list of owned Lego sets. Users can participate in auctions and place bids on them.

- **LegoSet**: Represents individual Lego sets with attributes such as name, description, and a list of photo media IDs. These sets are the primary items that can be auctioned.

- **Auction**: Represents an auction for a Lego set, with details such as the seller, base price, close date, and status. Auctions are linked to Lego sets and users.

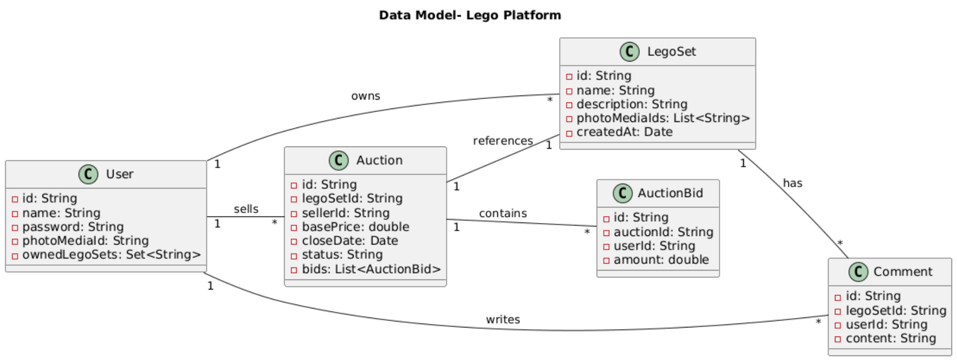
- **AuctionBid**: Captures bids placed on an auction. Each bid includes the auction ID, user ID, and the bid amount, and is associated with a specific auction.

- **Comment**: Allows users to add comments on Lego sets, providing feedback and interactions with the community.

These core data models are implemented through RESTful API endpoints that allow interaction with the system. Each model is designed to store and retrieve data efficiently, using Azure services like Cosmos DB for structured data and Blob Storage for media files.

2.2 UML Diagram

Below is a simplified UML diagram illustrating the relationships between these core components of the system. The diagram provides a visual representation of the classes and their interactions.



3 Services Used

The Lego platform backend leverages a set of key Azure services that are integral to its design and functionality. These cloud services provide the foundation for managing users, Lego sets, auctions, and bids, while ensuring scalability, reliability, and performance across the system. Each service plays a specific role in optimizing the backend architecture and supporting critical operations such as API hosting, data storage, and real-time interactions.

3.1 Azure App Service (PaaS)

Role: Primary hosting platform for the REST API backend (implemented in Java/JAX-RS).

- **PaaS (Platform as a Service):** Simplifies infrastructure management and operations, allowing developers to focus on business logic rather than server and infrastructure maintenance.
- **Scalability:** Azure App Service provides automatic scaling, adapting the number of instances based on traffic, which is crucial for handling peak loads during auctions.

3.2 Azure Cosmos DB

Role: Primary database for structured and transactional data (users, Lego sets, auctions, comments).

- **Low Latency:** Ensures fast read and write operations , critical for user experience and time-sensitive operations such as bidding on auctions.
- **Elastic Throughput (RU/s):** Dynamically allocates the required compute power to handle varying query volumes, crucial for absorbing the load generated by Artillery load testing scenarios.
- **NoSQL Model (Core/SQL API):** Provides flexibility for entity schemas and supports complex queries, such as filtering and sorting auctions.

3.3 Azure Blob Storage

Role: Storage for large, unstructured binary objects (e.g., Lego set photos, user profile pictures).

- **Offloading the Database:** Separates large data (media files) from transactional data, keeping Cosmos DB documents small. This ensures that RU/s are used efficiently for query and transaction processing rather than storing large image files.
- **Cost-Effectiveness:** A highly scalable, low-cost solution for storing large volumes of infrequently accessed data, like media files.

3.4 Azure Functions

Role: Handles asynchronous and event-driven business logic

- **Serverless:** Pay-per-use consumption, ideal for periodic tasks that don't require a continuously running server.

- **Specific Tasks (Cron Jobs):**
  - **Close Expired Auctions:** Triggered by a timer (Timer Trigger) to close auctions that have ended.
  - **Clean Older Auctions:** Manages the data lifecycle, removing very old auctions to maintain Cosmos DB performance.
  - **Analyze Comments:** Analyzes comments on Lego sets to determine if a Lego set is liked or not. This function is triggered by an HTTP request (HTTP Trigger) and uses sentiment analysis to assess the feedback.

3.5 Azure Cache for Redis

Role: Distributed in-memory cache for frequently accessed data.

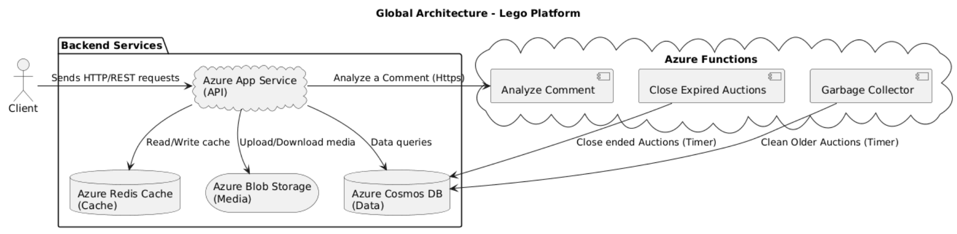
- **Reduced Latency:** GET requests are served in a few microseconds from the cache, significantly reducing API response times.
- **Offloading Cosmos DB:** Caches frequently read data, reducing the number of queries to Cosmos DB. This saves on RU/s consumption and frees up throughput for write operations and critical transactions (such as auctions).

3.6 Cache Strategy: Cache-Aside Pattern with Write Invalidation

Role: Implements a cache-aside strategy for handling frequently accessed data.

- **Cache-Aside Pattern:** The cache is populated only when data is requested, ensuring that only relevant data is stored in Redis. This reduces memory usage and ensures that the cache is up to date.
- **Write Invalidation:** When data is modified (such as a new bid in an auction), the cache is invalidated and updated with the new value, ensuring consistency between the cache and the database.

These services form a cohesive, cloud-native architecture, where each component serves a specific role while working together to deliver performance, scalability, and resilience. The combination of Azure's managed services (App Service, Cosmos DB, Functions, Redis) with the flexibility of Blob Storage and the efficiency of Tomcat ensures that the Lego platform can handle high traffic, large amounts of data, and provide a seamless experience to users while remaining cost-effective and reliable.



4 Implementation

The system follows a modular approach for scalability and maintainability. The API is designed using REST principles, and the interactions between different entities (users, Lego sets, auctions) are implemented as endpoints. The data is stored in Cosmos DB, and Blob Storage is used to handle large media files.

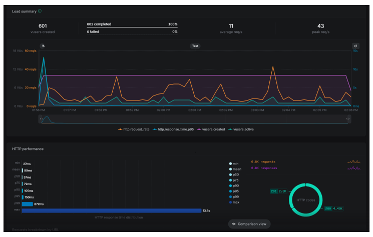
4.1 Key Endpoints

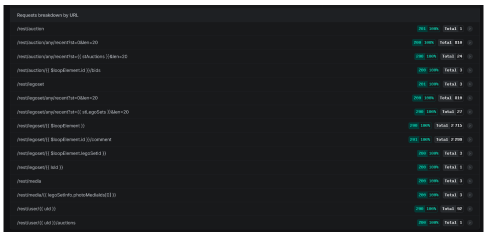
- `/rest/user`: Create, update, delete, retrieve, and list users.
- `/rest/media`: Upload and download media.
- `/rest/legoset`: Create, update, delete, retrieve, and list Lego sets.
- `/rest/auction`: Create and list auctions.
- `/rest/auction/id/bid`: Place bids on a given auction.

5 Evaluation

The system's performance is evaluated using Artillery to simulate different configurations:

- Application deployed in region France Central without caching.



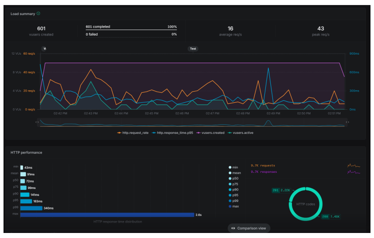


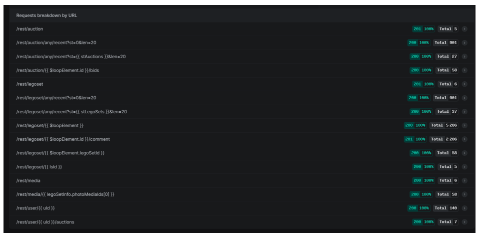
Requests breakdown by URL

/rest/auction	201	100%	Total	1
/rest/auction/any/recent?st=0&len=20	200	100%	Total	810
/rest/auction/any/recent?st={{ stAuctions }}&len=20	200	100%	Total	24
/rest/auction/{{ $loopElement.id }}/bids	200	100%	Total	3
/rest/legoset	201	100%	Total	3
/rest/legoset/any/recent?st=0&len=20	200	100%	Total	810
/rest/legoset/any/recent?st={{ stLegoSets }}&len=20	200	100%	Total	27
/rest/legoset/{{ $loopElement }}	200	100%	Total	2 715
/rest/legoset/{{ $loopElement.id }}/comment	201	100%	Total	2 299
/rest/legoset/{{ $loopElement.legoSetId }}	200	100%	Total	3
/rest/legoset/{{ lsId }}	200	100%	Total	1
/rest/media	200	100%	Total	3
/rest/media/{{ legoSetInfo.photoMediaIds[0] }}	200	100%	Total	3
/rest/user/{{ uId }}	200	100%	Total	92
/rest/user/{{ uId }}/auctions	200	100%	Total	1

Figure 1: Artillery Test without cache.

- Application deployed in region France Central with caching.




Requests breakdown by URL

/rest/auction	201 100%	Total 5
/rest/auction/any/recent?st=0&len=20	200 100%	Total 901
/rest/auction/any/recent?st={{ stAuctions }}&len=20	200 100%	Total 27
/rest/auction/{{ $loopElement.id }}/bids	200 100%	Total 58
/rest/legoset	201 100%	Total 6
/rest/legoset/any/recent?st=0&len=20	200 100%	Total 901
/rest/legoset/any/recent?st={{ stLegoSets }}&len=20	200 100%	Total 37
/rest/legoset/{{ $loopElement }}	200 100%	Total 5 286
/rest/legoset/{{ $loopElement.id }}/comment	201 100%	Total 2 206
/rest/legoset/{{ $loopElement.legoSetId }}	200 100%	Total 58
/rest/legoset/{{ lsId }}	200 100%	Total 5
/rest/media	200 100%	Total 6
/rest/media/{{ legoSetInfo.photoMediaIds[0] }}	200 100%	Total 58
/rest/user/{{ uId }}	200 100%	Total 140
/rest/user/{{ uId }}/auctions	200 100%	Total 7

Figure 2: Artillery Test without cache.

- Application deployed in a different region Y with or without caching. Due to the restrictions imposed by my home university on my Azure subscription, I was unable to deploy the backend in regions other than France Central. Azure's policy limits the regions available for resource deployment under this subscription, which impacted my ability to test the application in alternative regions. As a result, performance evaluations outside of France Central could not be performed, highlighting the dependency on a single region for the deployment and performance testing of the application.

6 Comparative Analysis With and Without Cache

The load testing results show a significant performance improvement with Redis cache enabled. The average response time drops from 99ms to 91ms, a reduction of **8%**. More importantly, the P99 percentile decreases from 872ms to 340ms, representing a **61%** improvement in stability. The previously high maximum response time of 13.8 seconds is reduced to 2.6 seconds, eliminating critical timeouts.

With caching, the performance distribution becomes more predictable. The gap between P99 (872ms) narrows 340ms, indicating that the system can handle extreme loads effectively. This improvement is due to Redis offloading repetitive read requests from Cosmos DB, allowing it to focus on write-heavy operations like bids. As a result, the system becomes more resilient, handling traffic spikes without direct database pressure.

Discussion: Beyond Performance

The implementation of caching goes beyond mere technical optimization—it plays a critical role in ensuring the stability and reliability of the system. Without caching, the application experienced variability in response times, with significant peaks that impacted performance under load. By using Redis, the platform is able to maintain consistent response times, even during high traffic, showcasing the importance of caching for improving system resilience. Caching in this context is essential for any system that needs to handle large amounts of traffic and maintain low-latency performance in a cloud environment.

7 Conclusions

This project demonstrates the implementation of a cloud-native, scalable backend architecture for a Lego platform using Azure services. The system effectively manages users, LegoSets, auctions, and comments while maintaining high availability and performance under load.

Key achievements include:

- A fully functional REST API built with Java/JAX-RS deployed on Azure App Service
- Efficient data management using Azure Cosmos DB with optimized query patterns
- Significant performance improvements through Azure Redis Cache, reducing P99 response times by 61% and eliminating critical timeouts
- Robust media handling via Azure Blob Storage with proper caching strategies
- Automated background processing using Azure Functions for auction management and system maintenance

The load testing results unequivocally validate our architectural choices, particularly highlighting that caching is not merely an optimization but a fundamental requirement for reliable cloud applications. The dramatic performance transformation—from unstable 13.8-second timeouts without cache to 2.6s with cache—demonstrates that Redis caching is essential for both performance and system stability. This project confirms that Azure's managed services provide a solid foundation for building applications, offering scalability, reliability, and cost-effectiveness. The implemented architecture serves as a blueprint for future cloud-native developments, proving that proper caching strategies and service selection are critical success factors in modern distributed systems.